27. Rich Components with A/P-Quality Contracts

1. CBSE for Embedded Systems
2. SPEEDS Heterogeneous Rich Components
3. Contract specification language CSL
4. Self-Adaptive Systems
5. HRC as Composition System

Many Slides are courtesy to Vered Gafni, Israel Aircraft Industries (IAI).
Used by permission for this lecture.
Other material stems from the SPEEDS project www.speeds.eu.com

Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de

Version 12-1.1, 27.06.12

Obligatory Literature

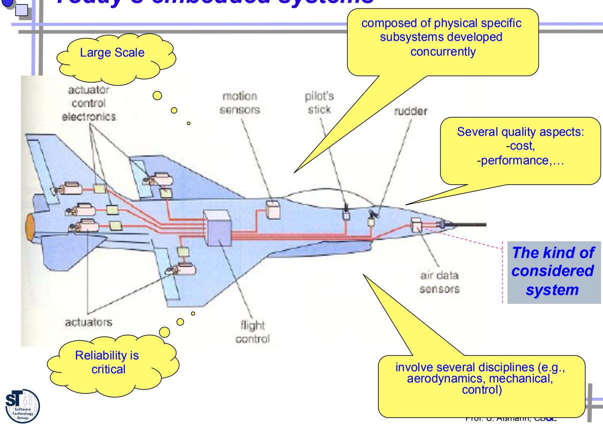
- www.speeds.eu.com
- G. Döhmen, SPEEDS Consortium. SPEEDS Methodology – a white paper. Airbus Germany.
  - http://www.speeds.eu.com/downloads/SPEEDS_WhitePaper.pdf
- [MM-Europe] R. Passerone, I. Ben Hafaiedh, S. Graf, A. Benveniste, D. Cancila, A. Cuccuru, S. Gerard, F. Terrier, W. Damm, A. Ferrari, A. Mangeruca, B. Josko, T. Peikenkamp, and A. L. Sangiovanni-Vincentelli. Metamodels in Europe: Languages, tools, and applications. IEEE Design & Test of Computers, 26(3):38-53, 2009.
- [Heinecke/Damm] H. Heinecke, W. Damm, B. Josko, A. Metzner, H. Kopetz, A. L. Sangiovanni-Vincentelli, and M. Di Natale. Software components for reliable automotive systems. In DATE, pages 549-554. IEEE, 2008.
- [Damm-HRC] Werner Damm. Controlling speculative design processes using rich component models. In Fifth International Conference on Application of Concurrency to System Design (ACSD'05), pages 118-119. IEEE Computer Society, 2005.

Used References

- [CSL] The SPEEDS Project. Contract Specification Language (CSL)
  - http://www.speeds.eu.com/downloads/ D_2_5_4_RE_Contract_Specification_Language.pdf
- [HRC-MM] The SPEEDS project. Deliverable D.2.1.5. SPEEDS L-1 Meta-Model, Revision: 1.0.1, 2009
  - http://speeds.eu.com/downloads/SPEEDS_Meta-Model.pdf
- [HRC-Kit] The SPEEDS project. SPEEDS Training Kit.
  - http://www.speeds.eu.com/downloads/Training_Kit_and_Report.zip
  - Training_Kit_and_Report.pdf: Overview
  - Contract-based System Design.pdf: Overview slide set
  - ADT Services Top level Users view.pdf: Slide set about different relatinoships between contracts
- G.Gößler and J.Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005.

27.1. CBSE for Embedded Systems

Today's embedded systems



Quality Requirements (Real-time, Safety, Energy, Dynamics)

- **Informal Quality Requirements** are specified in the *software requirements specification (SRS, Pflichtenheft)*
- **Informal Real-Time Requirement:** *The gate is closed when a train traverses the gate region, provided there is a minimal time distance of 40 seconds between two approaching trains.*
  - Hard Real-time: definite deadline specified after which system fails
  - Soft Real-time: deadline specified after which quality of system's delivery degradates
- **Informal Safety Requirement:** *If the robot's arm fails, the robot will still reach its power plug to recharge.*
- **Informal Energy Requirement:** *If the robot's energy sinks under 25% of the capacity of the battery, it will still reach its power plug to recharge.*
- **Informal Dynamic Movement Requirement:** *If the car's energy sinks under 5% of the capacity of the battery, it will still be able to break and stop.*

Vision: Modular Verification of Behavior of Embedded Systems

- Usually, Embedded Software is hand-made, verification is hard
- But fly-by-wire and drive-by-wire need verification
- Challenge 1: Quality requirements can be formalized and proven
  - How to formalize them?
  - How to prove them?
- Challenge 2: Proof can be computed in modules, proof is modular and can be reused as a proof component in another proof
  - Contracts serve this purpose: they prove assertions about components and subsystems
  - Whenever an implementation of a component is exchanged for a new variant, the new variant must be proven to be **conformant** to the old contract. Then the old global proof still holds
  - This is a CBSE challenge!

27.2. SPEEDS HRC (Heterogeneous Rich Components)

Rich Component Models

- Used for component-based software for embedded systems
- A **rich component** defines contracts in several *views* with regard to different *viewpoints*
  - A contract for functional behavior (functional view)
  - Several quality contracts, e.g.,
    - Real-time behavior (real-time view)
    - Energy consumption (energy view)
    - Safety modes (safety view)
    - Movements (dynamics view)
- The **contract** (about the observable behavior) of a component is described by state machines in the specific view (**interface automata**)
  - The interface automata encode infinite, regular path sets (traces)
  - They can be intersected, unioned, composed; they are decidable
  - Contracts can be proven
- Instead of an automaton in a contract, temporal logic can be used and compiled to automata (**temporal logic contract**)

Assumptions about Automata-Based Contracts

- A component has one thread of control
- A component is always in a finite set of states
- The behavior of a component can be described by a protocol automaton (interface automaton)
  - Compatibility is decidable
- A **hybrid automaton** is an automaton in which states and transitions can be annotated in different views
  - A **real-time automaton** is a hybrid automaton with real-time annotations
  - A **safety automaton** is a hybrid automaton with safety annotations
  - A **dynamics automaton** is a hybrid automaton with dynamics equations (physical movement, electricity movement)
  - An **energy automaton** is a hybrid automaton with energy consumption annotations

A/P Quality Contracts for CBSE

- [Gössler/Sifakis, Heinecke/Damm]
- **Composability** gives guarantees that a component property is preserved across composition/integration
- **Compositionality** deduces global semantic properties (of the composite, the composed system) from the properties of its components
- An **A/P-contract** is an if-then rule: under the assumption A, the component will deliver promise P (aka guarantee G)

Assertion — *Assertion*

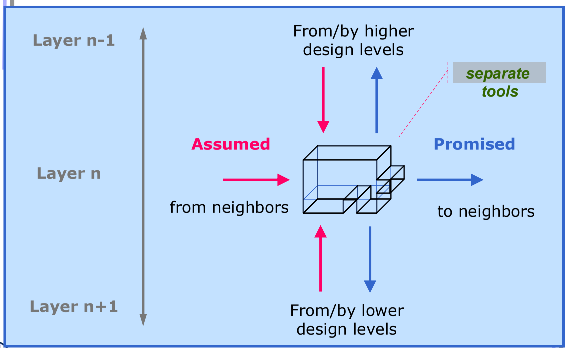
Contract = (**assumption**, **promise**)

= **IF assumption THEN promise**

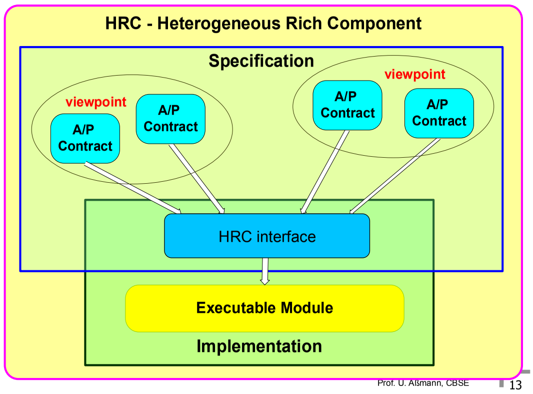
- An **A/P-quality contract** is an A/P-contract in which hybrid automata form the assumptions and promises

A/P-quality contract based component models are composable and compositional.

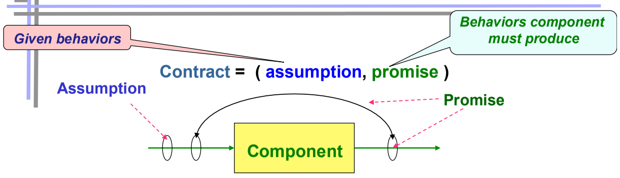
EU IP SPEEDS – Speculative and Exploratory Design in Systems Engineering



HRC – SPEEDS's View of a Component
An A/P-quality contract based component model



Basic Elements of HRC A/P-Contracts

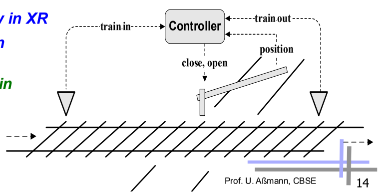


Assumption in natural language for a railway crossing XR:

- *Minimal delay of 50 sec. between successive trains*
- *At startup no train is already in XR*
- *Trains move in one direction*

Promise in natural language:

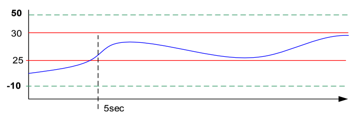
- *Gate closed as long as a train is in XR*
- *Gate open whenever XR is empty for more than 10 sec*



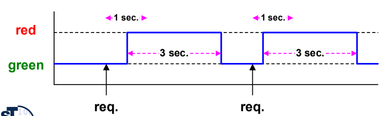
Assertions Describe Behavior

- An *assertion* specifies a subset of the possible component behaviors
- A finite automaton specifying an infinite set of regular paths

Contract = (**assumption**, **promise**)

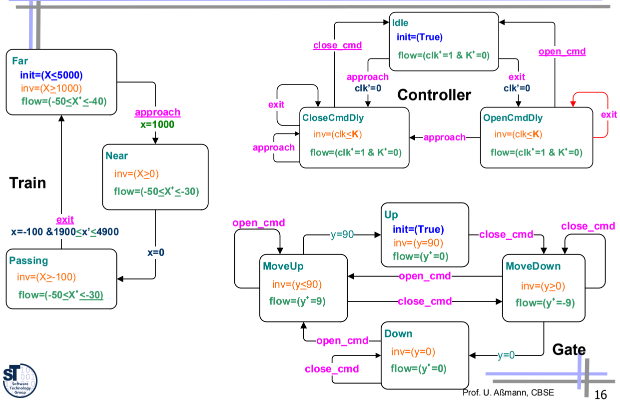


Contract over continuous variable:
temp: [-10°,50°]
'after 5 sec. 25≤temp≤30'

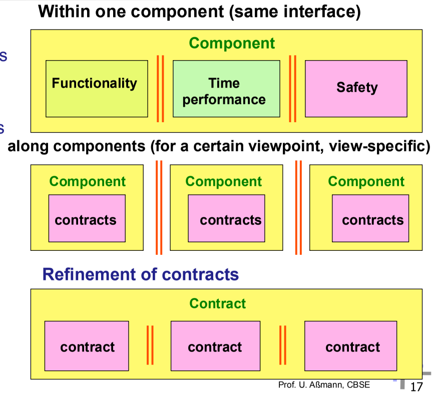


Contract over discrete variable:
lights :{*red*, *green*}, *req*: *event*
'lights initially green, and after each 'req', within 1sec, become red for 3 sec. then back green'

Hybrid Automata – Automata Representing Assertions



Contract Analysis



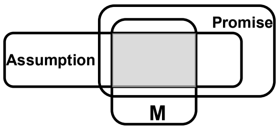
- is based on algebra of contracts
- For HRC contracts, the following properties can be proven:
  - Consistency,
  - Compatibility,
  - Dominance,
  - Simulation,
  - Satisfiability

Basic Relations on Contracts

- Satisfaction (implementation conformance) couples implementations to contracts.
- Given contract: C=(A,G), implementation *M*
- **Satisfaction: (M satisfies C)**

M|=C ⇔$_{def}$ A∩M⊆ G

(promise G is stronger than intersection of A and M)



Reasoning with Venn diagrams: smaller means weaker;
Inclusion means implication

Basic Relations on Contracts

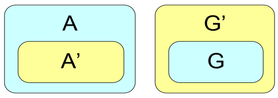
Given contract: C=(A,G) C'=(A',G'), *implementation M*

- ***Dominance: (C dominates C') :***

C<C'⇔$_{def}$ A'⊆A and G⊆G'

(A is stronger than A' and G' is stronger than G; A' is weaker than A and G is weaker than G')

contravariant in A and G, i.e, when assumption A "grows", the promise G "shrinks";



Example:

- ***C: A= daylight G= video & IR picture***
- ***C': A'= anytime G'= only IR picture***
- ***Daylight ⊆ anytime, video&IR picture ⊆ IR picture***

Claim: M|=C and C<C' ⇒ M|=C'

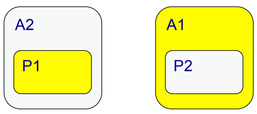
(if M satisfies C, and C dominates C', then M satisfies C')

Compatibility of Contracts

- Compatibility is a relation between two or more contracts *C1 .. Cn*
- Two contracts *C1* and *C2* are **compatible** whenever the promises of one guarantee that the assumptions of the other are satisfied
  - When composing their implementations, the assumptions will not be violated
  - The corresponding components "fit" well together
- *C1* = (*A1*,*P1*) and *C2* = (*A2*,*P2*) are compatible if

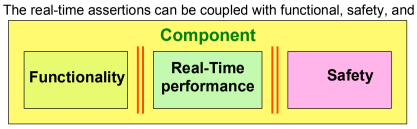
C1<->C2⇔$_{def}$ P1⊆A2 and P2⊆A1

C1 is compatible to C2 if C1.P is weaker than C2.A, and C2.P weaker than C1.A

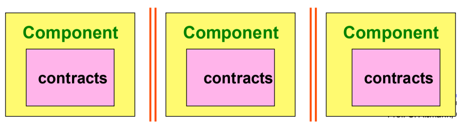


Composition of Contracts

- within a component (same interface), contracts in different views can be **synchronized**
  - The real-time assertions can be coupled with functional, safety, and energy view

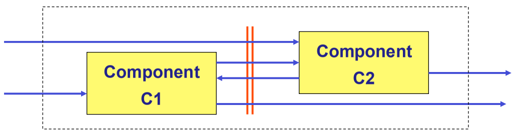


- along components – contracts of a certain viewpoint can be composed



Parallel Composition of Contracts (of separate components)

- Given contracts $C_1=(A_1,G_1)$, $C_2=(A_2,G_2)$, implementation *M*
- **Parallel composition** of contracts $C_1||C_2 = (A,G)$:=
- where: $A = (A1 \cap A2) \cup \neg(G_1 \cap G_2)$, $G = G_1 \cap G_2$

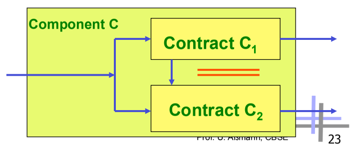


Algebra of Contracts

Given contracts $C_1=(A_1,G_1)$, $C_2=(A_2,G_2)$, the following operations can be defined:

- **Greatest Lower Bound:** $C_1 \sqcap C_2 =_{def} (A_1 \cup A_2, G_1 \cap G_2)$
- **Least Upper Bound:** $C_1 \sqcup C_2 =_{def} (A_1 \cap A_2, G_1 \cup G_2)$
- **Complement:** $\neg C =_{def} (\neg A, \neg G)$
- **Fusion:** $[[C1,C2]]_p = [C1]_p \sqcap [C2]_p \sqcap [C1||C2]_p$

$C=(A,G), p \in P \Rightarrow_{def} [C]_p = (\forall pA, \exists pG)$



Assertions Expression – Formal Language: Temporal Logic

- **In practice, Hybrid Automata are 'too formal' (too low level) to be used by normal engineers.**
  - **Alternative options like (Metric) LTL were examined and do better**

The gate is closed when a train traverses GR (gate region).

(EnterGR → ClosedUExitGR)

- **But for normal properties, logic is still too difficult and rejected by the engineers:**

P occurs within (Q,R)

((Q ∧ ¬R ∧ O¬R) ∧ ◊R) → (¬R)U(O(P ∧ ¬R)))

Between the time an elevator is called at a floor and the time it opens its doors at that floor the elevator can pass that floor at most twice.

((call ∧ ◊Open) → (Move U (Open ∨ (Pass U (Open ∨ (Move U (Open ∨ (Pass U (Open ∨ (Move U Open)))))))))

Assertions by Contract Patterns

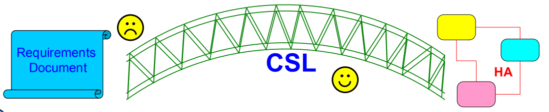
- A ***contract pattern (pattern rule)*** is an English-like template sentence embedded with parameters' placeholders, e.g.:
  inv [Q] while [P] after [N] steps

  represents a fixed property up to parameters' instantiation.
  (in the speak of the course, it is an English generic fragment of English)
- The semantics of a pattern is a template automaton (generic contract), which is instantiated by the parameters
  - A binding composition program translates the English sentence to a template automaton by binding its slots
- In the SafeAir project previous to SPEEDS, a contract patterns library was developed by OFFIS (Oldenburg), but the library grew up to ~400 patterns, and was not manageable
  - Parameters are instantiated by state expressions
  - Semantics over discrete time model

idea acceptable by users (format, less) but patterns can be very complex, like:
inv [P] triggers [Q] unless [S] within [B] after_reaching [R]

27.3 CSL (Contracts Specification Language) based on A/P-contract-patterns

- **CSL is a domain-specific language (DSL) intended to provide a friendly formal specification means**
  - **Translated into Hybrid Automata (assumptions and promises)**
  - **Template sentences from requirement specifications can be translated into interface automata**
- **CSL introduces events and time intervals in contract patterns**
- **CSL is a ECA language with real-time assertions**



CSL – Component Specification

- The CSL/HRC grammar defines interfaces with contracts of assumptions and promises.

HRC *{HRC-Id}*
Interface
controlled *{variables declaration}*
uncontrolled *{variables declaration}*
Contracts
{ *{viewpoint-id}* **contract** *{contract-id}* *
Assumption *{assertion}*
Promise *{assertion}* }

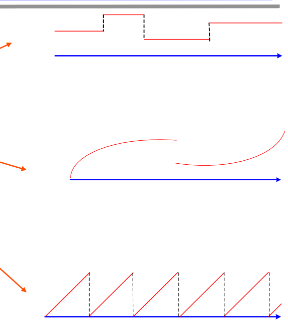
CSL Metamodel

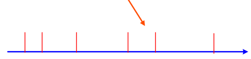
- [HRC-MM] is done in MOF and OCL
  - executable in MOF-IDE (Netbeans),
  - checked on well-formedness by OCL checkers
- Variables, assumptions
- More information about MOF-based metamodels and how to use them in tools -> Course Softwarewerkzeuge (WS)

{*viewpoint-id*} contract {*contract-id*}
Assumption: {*assertion*}*
Promise: {*assertion*}*

CSL Time model & variables

- **Time model: $R_{\geq 0}$.**
- **Variables:**
  Discrete range
  Continuous range
  pwc evolution
  ⇒ pw derivable
- **Events**





CSL – Contract Specification with Generic Text Fragments

- CSL uses generic programming for assertions

{*assertion*}: (text '[' slot:Parameter ']')*

- An ***assertion*** is expressed by a ***contract pattern***, a generic text fragment embedded with parameters (slots):
  - Parameter slots are ***conditions***, ***events***, ***intervals***.
  - Hedge symbols [ ] to demarcate slots

Example: Whenever the request button is pressed a car should arrive at the station within 3 minutes

***Whenever* [car-request] *occurs* [car-arrives] *occurs within* [3min]**

Contract Specification Process in HRC-CSL

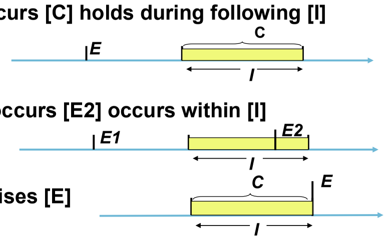
Steps to Derive Contracts:
- Start with the informal requirement
  - Identify what has to be guaranteed by the component under consideration and what cannot be controlled and hence should be guaranteed by the environment:
  - Informal promise(s), Informal assumption(s)
- Identify the related interface: inputs / outputs
- Specify parts of the informal requirements in terms of inputs and outputs of the component
- Select an appropriate contract pattern from the contract pattern library and substitute its parameter slots

Instantiation of a Contract Pattern

- *Whenever the request button is pressed a car should arrive at the station within 3 minutes*
- **Contract Pattern**:
  Whenever [E: event] occurs [E2: event] occurs within [I: interval]
- **Instantiated Contract**:
  Whenever *req-button-pressed* occurs *car-arrives-at-station* occurs within *3 min*
- Compiles to an hybrid automaton (here: real-time automaton)

More Contract Patterns

- whenever [E] occurs [C] holds during following [I]
- whenever [E1] occurs [E2] occurs within [I]
- [C] during [I] raises [E]



Temporal/Continuous expressions for parameters (Events, Conditions, Intervals)

Example: Formalization of Informal Requirement with a Contract Pattern

- Assertion:
  - Whenever the request button is pressed a car should arrives at the station within 3 minutes
- Instantiated in CSL:
  - *Whenever* [request-button-press] *occurs* [car-arrives-at-station] *holds within* [3min]

Contract with

- Assumption:
  - [40 seconds minimal delay between trains]
  - whenever [train_in] occurs [~train_in] holds during following (0,40]
- And Promise:
  - The gate is closed when a train traverses gate region.
  - [gate is closed when a train traverses gate region]
  - whenever [train_in] occurs [position==closed] holds during following [train_in, train_out]

Contract Pattern Parameters (Slots) and Their Typing

Conditions:

- **Boolean variables C**
- **x ~ exp** *-- K=8, x>5, y'= -3y² +7, x<y*
- **Exp.** $C_1 \vee C_2$ $C_1 \wedge C_2$ $\neg C$ $C_1 \rightarrow C_2$

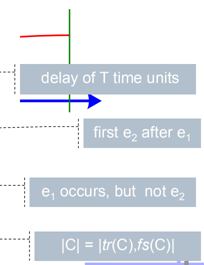
Events:

- **Primitive:** *a b c* *Startup*
- **Condition change:** *tr*(C) *fs*(C)
- **Time delay:** *dly*(T)
- **Exp.:** $e_1 \wedge e_2$, $e_1 \vee e_2$, e_1-e_2, *e when C* $e_1;e_2$

Intervals:

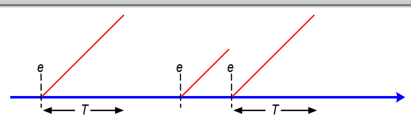
- **Designated by occurrences of events, a, b; all forms:**
  [a,b], [a,b), (a,b], (a,b)

A condition must hold true along an interval



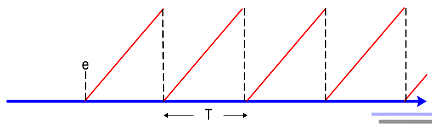
Timers

Timer(T) at e



- e+T ≡ *tr*(c=T) where c=Timer(T) at e

PeriodicTimer(T) at e



CSL Examples with Timers

Dispatching commands will be refused during first 5 seconds after a car arrives at station

- *Whenever* [car-arrives] *occurs* [dispatch-cmd] *implies* [refuse-msg] *during following* [5sec]

40 sec. minimal delay between trains:

- *Whenever* [Tin] *occurs* [Tin] *does not occur during following* (40 sec]

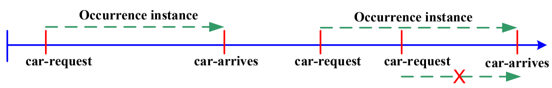
Between the time an elevator is called at a floor and the time it stops at that floor the elevator can pass that floor at most twice.

- [PassFloor[m]] *occurs at most* [2] *times* *during* (CallAtFloor[m], StopAtFloor[m])

Pattern Occurrence Types

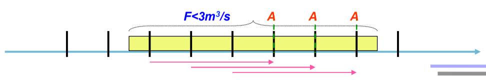
Iterative occurrences of events – non interleaving occurrence's instances

Whenever [car-request] *occurs* [car-arrives] *occurs within* [3min]



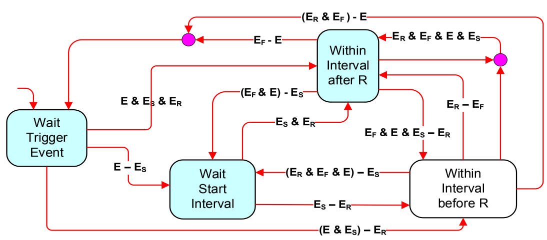
Flowing occurrences of events - interleaving occurrence's instances

[F<3] during [3 Sec] raises [AlarmSignal]



Automaton Representation of Iterative Occurences of Events

whenever [E] occurs [E_R] occurs within [E_S,E_F]



More HRC Patterns for Contract Specification

- E: Event, SC: State Condition, I: Interval, N: integer
- **Pattern Group "Validity over Duration"**
- **P1 (hold)**: whenever [E] occurs [SC] holds during following [I]
- **P2 (implication)**: whenever [E1] occurs [E2] implies [E3] during following [I]
- **P3 (absence)**: whenever [E1] occurs [E2] does not occur during following [I]
- **P4 (implication)**: whenever [E] occurs [E/SC] occurs within [I]
- **P5**: [SC] during [I] raises [E]
- **P6**: [E1] occurs [N] times during [I] raises [E2]
- **P7**: [E] occurs at most [N] times during [I]
- **P8**: [SC] during [I] implies [SC1] during [I1] then [SC2] during [I2]

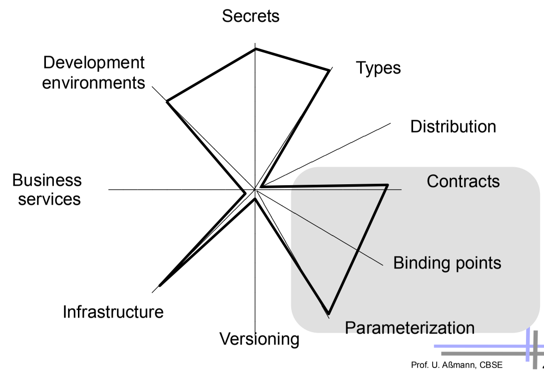
27.4. Self-Adaptive Systems

- For future networked embedded systems and cyber-physical systems, we need **verifiable, compositional** component models supporting **self-adaptivity**.
- Self-adaptivity can be achieved by dynamic product families with variants that are preconfigured, verified, and dynamically reconfigured:
  - **Contract negotiation** (dynamic reconfiguration between quality A/P-automata)
  - Polymorphic classes with **quality-based polymorphism**: the polymorphic dispatch relies on quality types, quality predicates
  - **Autotuning** with code rewriting and optimization
- More in research projects at the Chair

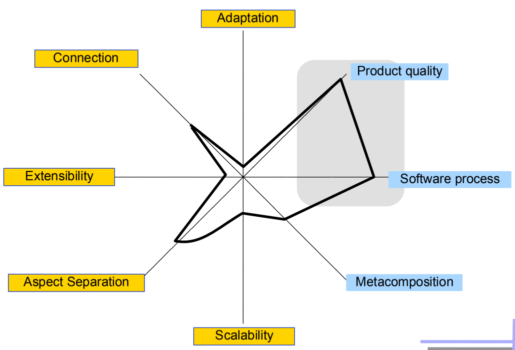
27.5 HRC as Composition System

- HRC is an interesting combination of a black-box component model in *different views*
- It could be one of the first COTS component models with viewpoints, but the standarization is unclear at the moment

Evaluation of HRC Component Model



HRC – Composition Technique and Language



HRC as Composition System